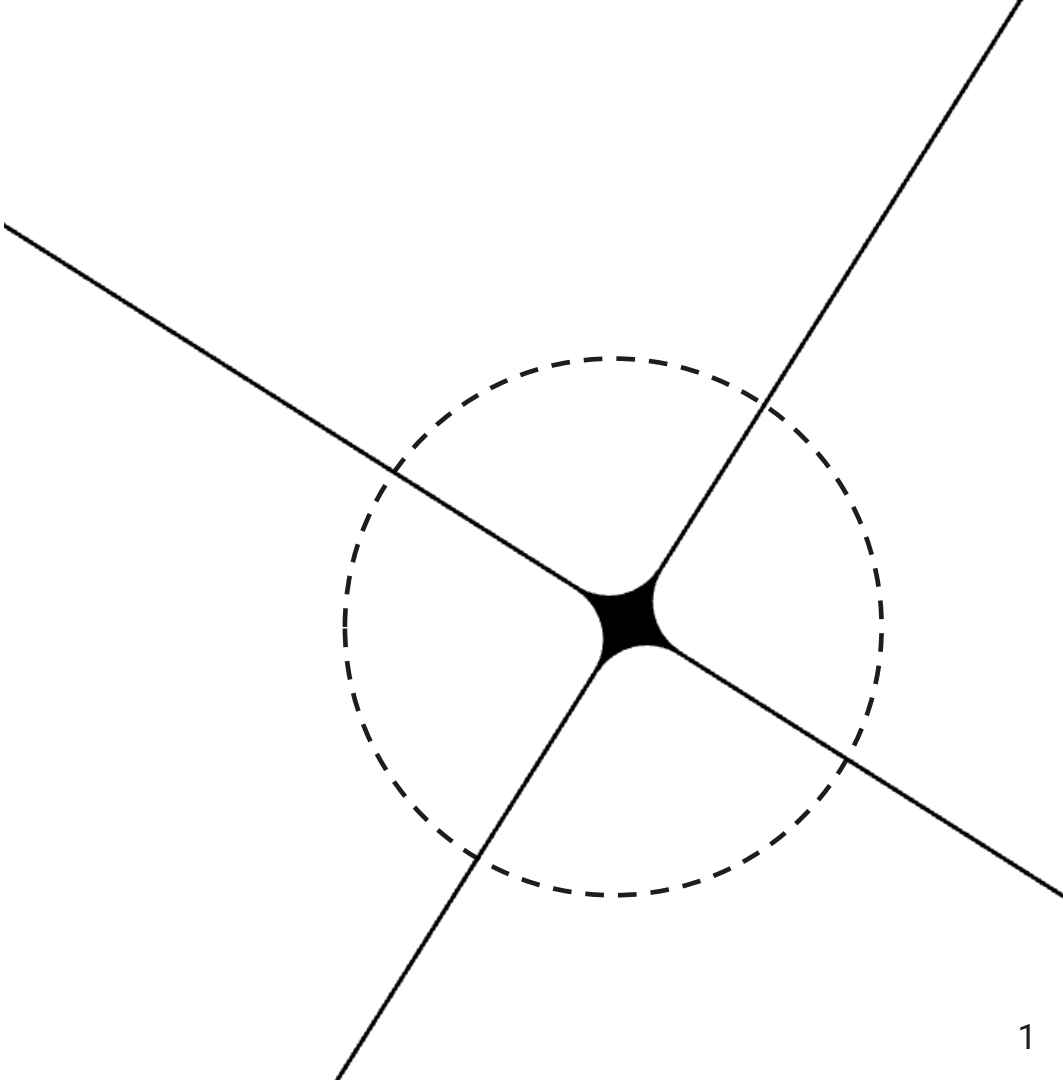
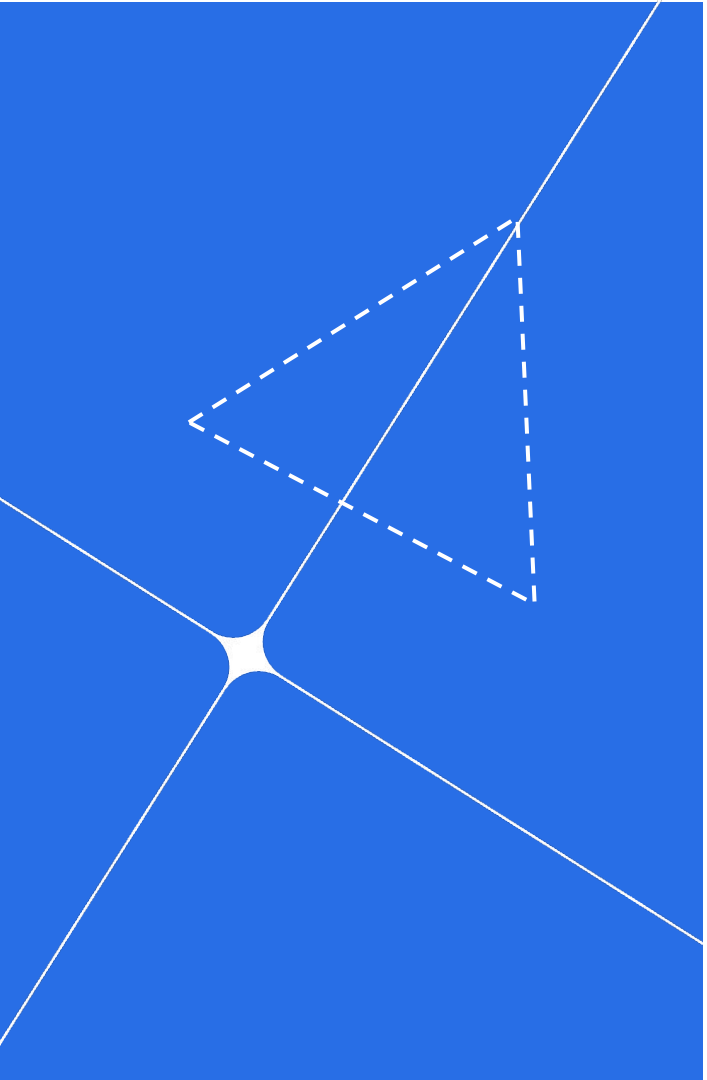
INSTITUT POLYTECHNIQUE DE PARIS

TELECOM SudParis

Feature stores & Feast

Julien Romero





Motivation & Concepts du Feature Store

Pourquoi un Feature Store ?

- Les features sont au cœur de la performance d'un modèle.
- Sans standardisation : calculs incohérents, duplication, bugs silencieux.
- Les équipes doivent partager, versionner et réutiliser des features.
- En production : besoin d'accéder aux features **rapidement, systématiquement, sans recalculer**.
- Le Feature Store fournit un cadre structuré : définition, stockage, serving, et historique des features.

Limites du calcul de features “à la main”

- Scripts dispersés entre notebooks, API, scripts d’entraînement.
- Logique de transformation dupliquée, parfois contradictoire.
- Mises à jour manuelles source d’erreurs.
- Difficile d’assurer que l’API reproduit exactement la logique du training.
- Aucune garantie temporelle : possible fuite d’information.
- Peu ou pas de réutilisation cross-projets.

Training-Serving Skew : problème central

- **Définition**: divergence entre les features utilisées à l'entraînement et celles calculées pour l'inférence.
- Un des principaux facteurs d'échec d'un système ML en production.
- Peut rendre un modèle performant "offline" mais catastrophique en conditions réelles.
- Origine : absence d'un mécanisme commun de définition/serving des features.

Causes du skew en pratique

- Code de feature engineering réécrit dans l'API au lieu d'être partagé.
- Agrégations calculées différemment selon l'environnement.
- Pipelines training et inference non synchronisés.
- Données plus fraîches en production qu'à l'entraînement (ou inverse).
- Différences de type, schéma ou mapping lors de l'ingestion.
- Bugs silencieux liés à des corrections appliquées après coup sur l'historique.

Conséquences : dérive silencieuse, bugs coûteux

- Baisse soudaine ou progressive de la précision du modèle.
- Prédictions instables ou incohérentes selon les utilisateurs.
- Difficulté extrême de diagnostic : “le modèle est-il mauvais ou les features sont-elles incohérentes ?”
- Coûts opérationnels élevés : réentraînements inutiles, hotfix API, rollback de modèles.
- Risque business réel : churn mal détecté, pertes de revenus.

Objectifs d'un Feature Store moderne

- **Définition centralisée** des features (déclarative, versionnée).
- **Reproductibilité** exacte entre offline et online.
- **Point-in-time correctness** pour éviter la fuite de données.
- **Serving** efficace :
  - Offline (training, batch scoring).
  - Online (latence faible pour l'inférence).
- **Partage** : plusieurs modèles/systèmes peuvent réutiliser les mêmes features.
- **Traçabilité complète** : source, schéma, timestamps, versions.

Offline vs Online Features

- **Offline**
  - Utilisés pour : entraînement, validation, backtesting.
  - Données historiques, volumineuses.
  - Agrégations lourdes autorisées.
  - Temporellement correctes via snapshots ou stores dédiés.
- **Online**
  - Utilisés pour : prédiction en temps réel.
  - Latence très faible exigée (<10–50 ms).
  - Taille plus limitée.
  - Doivent être synchronisés avec les features offline.

Temps réel vs batch : contraintes opérationnelles

- **Batch (offline) :**
  - Fenêtre temporelle large (ex : 30j, 90j).
  - Processus intensifs (ingestion, agrégations).
  - Pas besoin de retour immédiat.
- **Temps réel (online) :**
  - Doit répondre instantanément aux requêtes de l'API.
  - Matérialisation nécessaire : pas de recalcul dynamique.
  - Forte contrainte de cohérence avec la logique offline.

Versionnage des features et traçabilité

- Les features évoluent : définition, fenêtres temporelles, agrégations.
- Besoin de garder :
  - Version de chaque Feature View.
  - Mapping vers la source (snapshots).
  - Paramètres utilisés lors de l'entraînement.
- Sans versionnage : impossible de reproduire un modèle ou comprendre une dérive.
- Le Feature Store est un **métastore** des features.

Point-in-Time Correctness : intuition

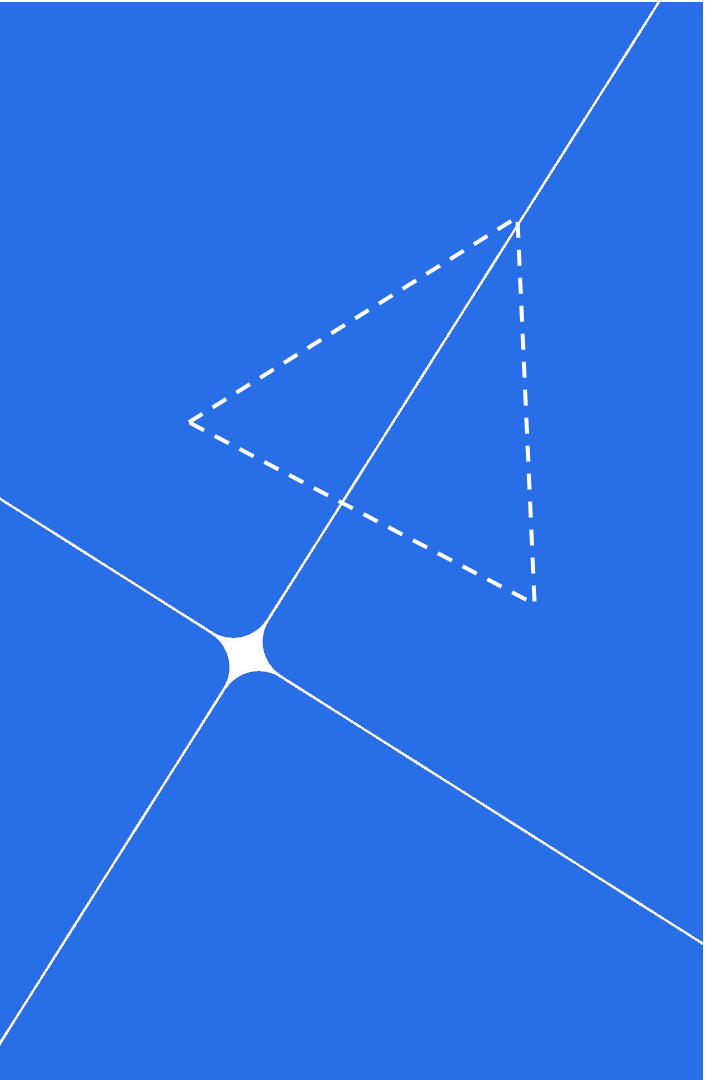
- Problème : un modèle peut “voir le futur” s'il utilise des données postérieures au label.
- Principe :
  - Pour chaque observation, seules les features **disponibles à cette date** sont utilisées.
  - Alignement strict entre timestamp du label et timestamp des features.
- Le Feature Store assure automatiquement cette garantie via son moteur de jointure temporelle (historical retrieval).

Comparaison : sans FS vs avec FS

- Sans Feature Store
  - Du code partout.
  - Skew fréquent.
  - Peu de réutilisation.
  - Pas de versionnage clair.
  - Tests limités, difficilement automatisables.
  - Diagnostique très difficile.
- Avec Feature Store
  - Définition unique et centralisée.
  - Alignement automatique training/inference.
  - Meilleure robustesse opérationnelle.
  - Historique et traçabilité garantis.
  - Rapidité à développer de nouveaux modèles.

Intégration dans notre architecture StreamFlow

- Ingestion + validation (TP1–TP2) produisent des **snapshots mensuels**.
- Feast consomme ces snapshots comme **offline store**.
- Développeurs définissent des **entities, data sources, feature views**.
- Matérialisation vers un **online store** pour l'API FastAPI.
- Entraînement (Lecture 4) récupérera un dataset cohérent via historical retrieval.
- Serving : FastAPI interroge Feast, puis le model, et renvoie la réponse.



Architecture de Feast

Position de Feast dans le pipeline MLOps

- Feast intervient entre l'ingestion/snapshots (TP1–TP2) et l'entraînement/modèle (TP4).
- Rôle : fournir un accès **cohérent, versionné, temporellement correct** aux features.
- C'est la couche intermédiaire qui garantit :
  - Alignement training/inference
  - Réutilisation de transformations
  - Découplage entre data engineering et modèle
- Fait le pont entre :
  - PostgreSQL (données structurées) ⇔ Training pipeline ⇔ API en production

Vue d'ensemble : repo -> registry -> offline/online stores

- **Feast repo** : répertoire déclaratif contenant la définition des features.
- **Registry** : base interne stockant la configuration versionnée (entities, FV…).
- **Offline Store** : source historique (PostgreSQL dans notre cas).
- **Online Store** : stockage rapide (Redis ou SQLite) pour l'inférence temps réel.
- Flux standard :
    1. Définir features
    2. feast apply (synchronise registry)
    3. Materialize offline puis online
    4. Serve features à l'API et au training.

Concept 1 : Entities

- Une **Entity** identifie l'unité principale pour laquelle on calcule des features.
  - Exemples : user, device, transaction, produit.
  - Dans notre projet : user.
- Propriétés essentielles :
  - nom unique
  - clé primaire (user_id)
    - stable dans le temps
    - correspondant aux clefs dans PostgreSQL
  - type (int, string, UUID…)
- Sert de pivot pour toutes les jointures temporelles.
- Pseudo-Python (exemple) :
  - ```
    user = Entity(name="user", join_keys=["user_id"], description="Identifie un utilisateur StreamFlow")
    ```
    ```

Concept 2 : Data Sources (PostgreSQL dans le TP)

- Une **Data Source** décrit où les features brutes/historiques sont stockées.
  - Dans le TP : tables snapshot dans PostgreSQL.
- Contient :
  - connexion DB
  - table ou requête SQL
  - timestamp field (as_of)
    - récupération historique
    - alignement avec les labels
    - exclusion automatique des données postérieures (anti-data leakage)
  - mapping des champs
- Importance : la validité temporelle dépend de cette source.
- Pseudo-Python :

```
PostgresSource(
    table="subscriptions_profile_snapshots",
    timestamp_field="as_of"
)
```

Concept 3 : Feature Views

- Un Feature View regroupe les features associées à une entity.
- Contient :
  - entities target : liste d'Entities liées
  - data source : table snapshot (PostgresSource ici)
  - liste des features (nom + type)
  - timestamp
  - TTL : période de validité en online store
  - tags : pour la documentation
- Acte central : déclarer les features plutôt que les calculer à la main.
- Correspond entièrement aux tables snapshot que nous avons construites.
- Pseudo-Python :

```
subscriptions_fv = FeatureView(
    name="subscriptions_profile",
    entities=["user"],
    schema=[Field("plan_type", String), Field("status", Int64)],
    source=subscriptions_source
)
```

Best practices : granularité des FeatureViews

- On ne fait pas un FeatureView par un modèle. Un FV = un bloc logique homogène.
- Éviter :
  - un FV gigantesque avec 100+ features
  - fusionner des domaines hétérogènes (ex : paiements + support)
- Favoriser :
  - regroupements naturels (durées, fenêtres temporelles)
  - modularité pour réutiliser des FV dans plusieurs modèles

Best practices : stabilité de schéma (liée à GE)

- Les FeatureViews supposent un schéma stable, d'où validation GE en amont.
- Modifier un FV :
  - ajouter feature = OK
  - supprimer ou renommer = versionner
- Feast ne valide pas la qualité des données : GE reste indispensable.

Concept 4 : Feature Service

- Un Feature Service regroupe plusieurs FeatureViews logiquement liés.
- Permet d'exposer un ensemble cohérent de features pour :
  - un modèle spécifique (ex : modèle churn)
  - un endpoint API
- Abstraction clé : un modèle ne dépend pas de la structure interne du Feature Store.
- Pseudo-Python :

```
churn_service = FeatureService(
    name="churn_service",
    features=[subscriptions_fv, usage_fv, payments_fv]
)
```

Concept 5 : Offline Store (training et historique)

- L’offline store est utilisé pour :
  - récupération historique (training)
  - backtesting
  - génération de datasets volumineux
- Dans le TP : PostgreSQL, alimenté par nos snapshots mensuels.
- Feast exécute automatiquement les point-in-time joins entre labels et features.
- Pas de recalcul lourd dans le code d’entraînement : Feast gère la reconstruction du dataset.

Concept 6 : Online Store (inférence)

- L’online store sert les features pour l’API en temps réel.
- Caractéristiques :
  - faible latence, lookup direct
  - stockage clé/valeur basé sur entity_id + timestamp
  - contenu mis à jour via la matérialisation
- Dans le TP : petit online store local.
- Permet à FastAPI de récupérer 10-20 features en quelques millisecondes.

Concept 7 : Materialization (offline -> online)

- Étape centrale : transférer les données snapshot vers le online store.
- Effectuée via :
  - `feast materialize <start> <end>`
- Permet d’éviter les recalculs au moment de l’inférence.
- Assure :
  - faible latence
  - cohérence avec les données historiques
  - refresh programmé (ex : quotidien, mensuel)
- Dans notre TP : matérialisation mensuelle basée sur les snapshots.

Cycle de vie d'un feature dans Feast

1. Ingestion & validation dans PostgreSQL (TP1–TP2).
2. Création d'un snapshot mensuel (as_of).
3. Déclaration du FeatureView (schema + source).
4. feast apply => mise à jour du registry.
5. Materialize => remplissage du online store.
6. Offline retrieval => construction du dataset d'entraînement.
7. Online retrieval -> FastAPI -> modèle -> prédiction.

Cycle entièrement reproductible et versionné.

Utilisation de feast apply

- Commande clé :
  - feast apply
- Effets :
  - synchronise le registry avec les FV déclarés
  - détecte les ajouts/modifications
  - avertit si des changements cassent la compatibilité
- Exécution à chaque changement de configuration.

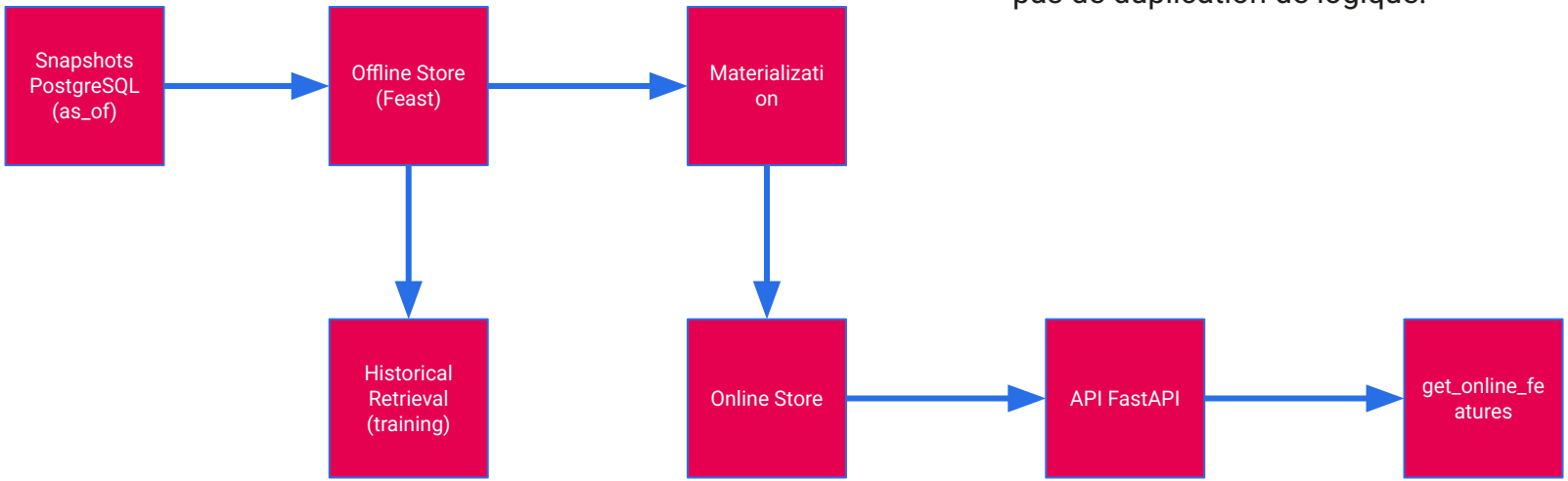
Fichiers du repo Feast dans le projet

- Dans services/feast_repo/repo :
  - feature_store.yaml
    - configuration globale (offline store, online store).
  - entities.py
    - définition des entities.
  - data_sources.py
    - configuration des tables PostgreSQL.
  - feature_views.py
    - un fichier par FeatureView (une seule ici)
  - services.py
    - Feature Services (groupement logique).
  - Registry (généré après feast apply).
- L'ensemble constitue la spécification déclarative des features.

Résultat : un registry versionné

- Après feast apply, Feast génère :
  - un registry.db (ou fichier équivalent)
  - contenant toutes les versions des FV, Entities, Sources
- Un registry :
  - permet la reproductibilité des entraînements
  - garantit que l'API et le pipeline training utilisent la même définition
- Le Feature Store devient la vérité unique des features du système.

Flux : snapshots -> Feast offline -> retrieval

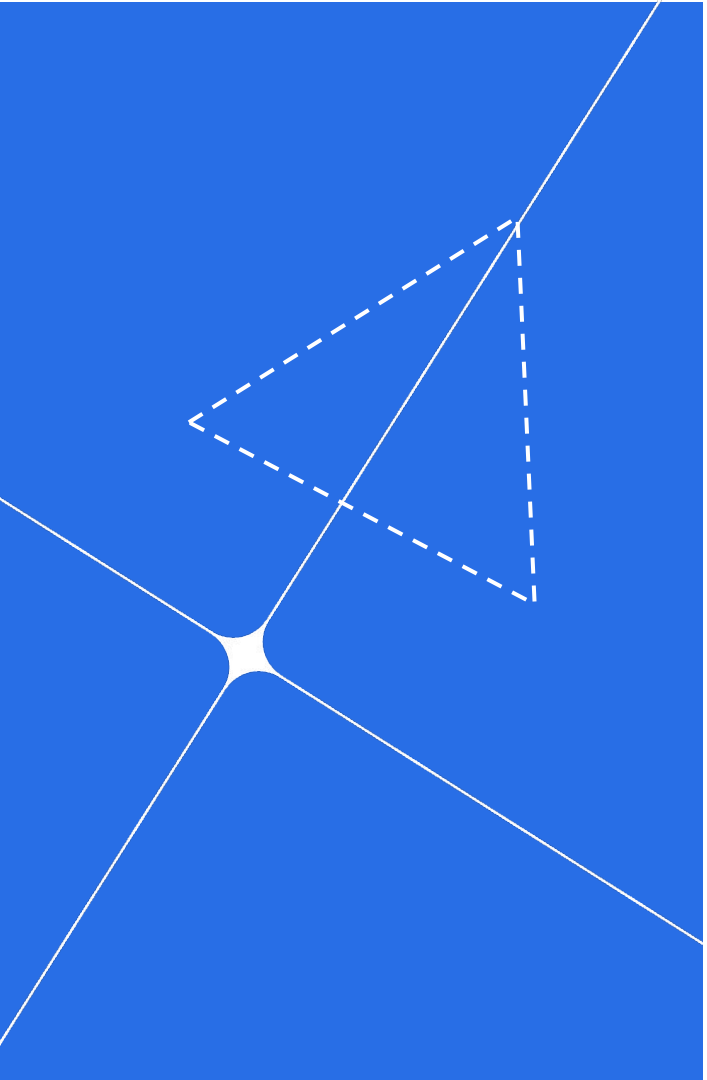


Points importants :

- Un seul pipeline de features, deux modes de consommation.
- Toujours basé sur les mêmes définitions => pas de duplication de logique.

Pourquoi Feast résout le skew

- Une **seule déclaration** de features utilisée partout.
- Serving offline/online basé sur la **même source**, garantissant cohérence.
- Jointures temporelles automatiques, impossibles à maintenir manuellement.
- Versionnage du registry → reproductibilité du modèle.
- Matérialisation contrôlée pour garantir la fraîcheur.
- Découplage : data engineers, ML engineers, API engineers utilisent la même base fonctionnelle.
- Le modèle voit exactement les mêmes features en entraînement et en production.



Offline Feature Retrieval (Training + Temporal Correctness)

Objectif : reconstruire un jeu d'entraînement

- Pour entraîner un modèle, il faut un dataset contenant :
  - un label (ex : churn / non churn)
  - un timestamp correspondant à la date du label
  - Toutes les features valides à ce moment-là
- Le rôle de Feast :
  - reconstruire automatiquement les features telles qu'elles existaient à une date passée
  - garantir une cohérence temporelle et structurelle
- L'ensemble forme un dataset utilisable pour un entraînement reproductible.

Labels + timestamp = clé d'alignement

- Chaque label est associé à :
  - un user_id
  - un label_value (ex : churn = 1)
  - un label_timestamp (date de référence)
- Le Feature Store utilise ce timestamp pour :
  - chercher les features correspondantes à cette date
  - Ignorer toute donnée future
- Alignement essentiel pour éviter le biais de fuite de données.

Rôle du timestamp dans l'historisation

- Les snapshots construits dans TP2 forment un historique mensuel.
- Chaque ligne de feature est datée par as_of.
- Feast reconstruit l'historique complet en choisissant :
  - le snapshot le plus récent avant le timestamp du label
- Cela permet de :
  - refaire un entraînement identique plusieurs mois ou années plus tard
  - auditer une prédiction passée

Point-in-time join

- Pour chaque (user_id, label_timestamp) :
  - sélectionner dans les snapshots les features valides au moment du label
- Condition :
  - feature_timestamp <= label_timestamp
- Et choisir la plus récente :
  - ORDER BY feature_timestamp DESC LIMIT 1
- Feast automatise cela : aucune jointure SQL manuelle nécessaire.

Pseudo-Python : get_historical_features

```
from feast import FeatureService

training_df = store.get_historical_features(
    entity_df=labels_df,         # contient user_id + event_timestamp + label
    features=FeatureService("churn_service")
).to_df()
```

- entity_df = DataFrame contenant user_id + event_timestamp (timestamp du label) + label.
- Feast renvoie un DataFrame fusionné :
  - user_id, label, et toutes les features correctes à la date donnée.

Exemple : jointure snapshots + labels

- Labels :
  - user_id = 42, churn = 1, event_timestamp = 2024-02-05
- Snapshots disponibles :
  - as_of = 2024-01-31
  - as_of = 2024-02-29
- Résultat du retrieval :
  - Feast prend 2024-01-31, car c'est le dernier snapshot avant 2024-02-05.
- L'utilisateur ne voit jamais cette logique.
- Feast applique automatiquement la règle temporelle.

Gestion des valeurs manquantes / users inconnus

- Users absents dans certains FeatureViews :
  - Feast renvoie NULL (ou NaN) => à gérer dans le pipeline de training.
- Features manquantes dans certaines fenêtres temporelles :
  - typique des nouveaux utilisateurs
  - acceptable tant que le modèle sait gérer ces cas
- Users totalement inconnus :
  - Feast renvoie une ligne vide => peut être filtré en prétraitement.
- Bonnes pratiques :
  - appliquer des imputations simples (0, médiane, catégorie “unknown”) dans le pipeline ML.

Résultat : un dataset cohérent et reproductible

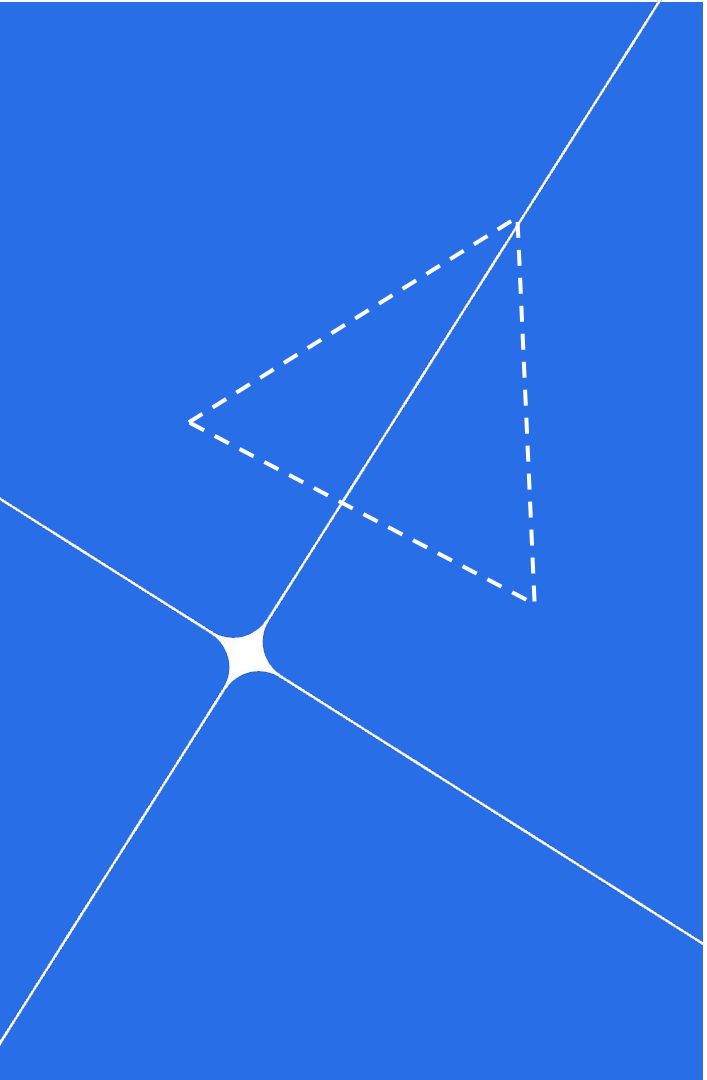
- Le retrieval garantit que :
  - toutes les features proviennent de la même définition versionnée (registry).
  - aucun calcul n'est refait manuellement.
  - aucune information future n'est injectée.
  - si l'ingénierie de features change, alors un nouveau registry versionné est utilisé.
  - le dataset est recréable à l'identique dans 6 mois.
- C'est le fondement de la reproductibilité en MLOps.

Préparation pour l'entraînement

- Le résultat du offline retrieval :
  - DataFrame complet : labels + features + user_id
  - ordres de colonnes uniformes
  - dataset prêt à être :
    - nettoyé (imputation, normalisation)
    - splitté (train/test/validation)
    - Loggé dans MLflow pour traçabilité
  - Le cours 4 montrera :
    - l'entraînement du modèle
    - l'enregistrement dans MLflow
    - le passage en production

Lien avec la future API (FastAPI + online store)

- L’API en production appellera :
  - store.get_online_features(feature_service, {"user_id": X})
- Les features récupérées online doivent correspondre exactement à celles utilisées offline.
- Grâce à :
  - la définition unique des FeatureViews
  - la matérialisation régulière du online store
- Conséquence :
  - Pas de recalcul manuel dans l’API
  - Plus de training-serving skew
  - Modèle robuste et cohérent



Online Features & Materialization

Pourquoi un Online Store ?

- Le modèle en production doit accéder aux features en temps réel.
- Impossible de recalculer des agrégations complexes à chaque requête.
- Le Online Store fournit :
  - lookup très rapide (ms) basé sur user_id
  - features déjà préparées via la matérialisation
  - cohérence avec l'historique (même définition que l'offline)
- C'est un cache structuré, versionné, optimisé pour l'inférence.

Matérialisation périodique : offline vers online

- Étape consistant à pousser les snapshots (offline store) dans le online store.
- Utilisée pour :
  - préparer les features avant les requêtes de l'API
  - synchroniser offline et online
  - garantir que le modèle en production utilise les données les plus récentes
- Exécutée :
  - manuellement
  - automatiquement (cron, orchestrateur Prefect dans les cours suivants)
  - Périodicité dépend du système (mensuelle dans notre TP).

Pseudo-Python : materialize(start, end)

```python
from feast import FeatureStore

store = FeatureStore(repo_path=".")

store.materialize(
    start_date="2024-01-01",
    end_date="2024-02-01"
)
```

- Feast récupère toutes les lignes offline entre start_date et end_date.
- Les insère dans l'online store sous forme de clés (entity_id => features).
- Aucun recalcul : simplement un transfert.

Structure du Online Store dans notre TP

- Store local, léger :
  - Redis (optionnel selon config)
  - SQLite (par défaut en mode local)
  - PostgreSQL dans le TP
- Organisation clé-valeur :
  - key = (entity, user_id)
  - value = { "feature_1": ..., "feature_2": ..., ... }
- Stockage déterminé par :
  - FeatureViews déclarés
  - TTL éventuel
- Vidange ou refresh déclenché par de nouvelles matérialisations.

Online retrieval : get_online_features

```
features = store.get_online_features(
    feature_service="churn_service",
    entity_rows=[{"user_id": 42}]
).to_dict()
```

Retourne un dictionnaire :

```
{
  "nb_sessions_30d": 12,
  "total_paid_90d": 29.99,
  "status": 1,
  ...
}
```

Appelé directement dans l'API FastAPI.

Latence extrêmement faible.

Exemple d'accès en temps réel (pseudo-API)

```python
@app.post("/predict")
def predict(request: PredictRequest):
    features = store.get_online_features(
        "churn_service",
        [{"user_id": request.user_id}]
    ).to_dict()

    model = mlflow.pyfunc.load_model("models:/churn_model/Production")
    y_pred = model.predict(features)

    return {"prediction": float(y_pred)}
```

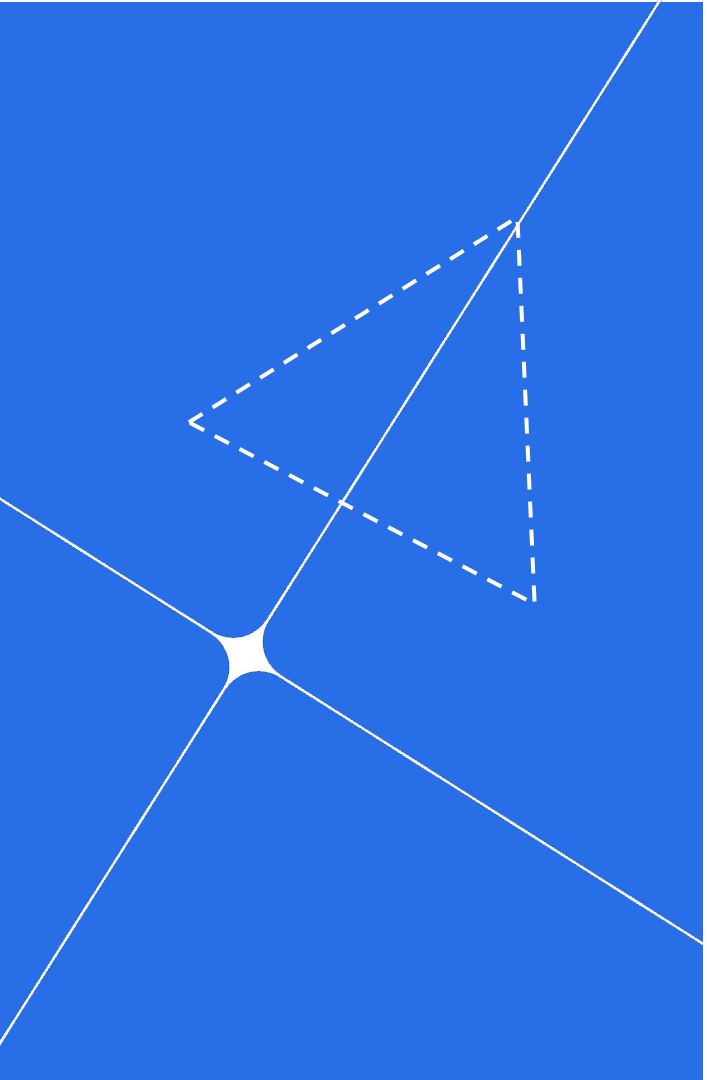
- Aucune transformation métier dans l'API.
- Le Feature Store fournit déjà les valeurs prêtes.
- Assure rigidité et cohérence.

Contrainte de fraîcheur (“freshness”)

- Les features online doivent être :
  - suffisamment fraîches pour être pertinentes
  - mais pas recalculées à chaque requête
- Paramètres influençant la fraîcheur :
  - fenêtre de matérialisation
  - fréquence des mises à jour
  - TTL éventuel sur certaines FeatureViews
- Dans le TP : données mensuelles, fraîcheur = “dernier snapshot disponible”.

Connexion avec le futur modèle servi

- Le modèle servira les prédictions via FastAPI + online store.
- Conséquences de Feast :
  - même définition des features utilisée en entraînement et en production
  - réduction drastique du risque de training-serving skew
  - pipeline API minimal et propre
- Étape suivante (Lecture 4) :
  - entraîner un modèle
  - l'enregistrer dans MLflow
  - connecter FastAPI à MLflow + Feast



Introduction TP 3

Objectifs du TP 3

- À la fin du TP, vous saurez :
  - Définir une Entity pour l'utilisateur.
  - Déclarer des DataSources PostgreSQL basées sur les snapshots (TP2).
  - Créer plusieurs FeatureViews cohérentes.
  - Appliquer la configuration avec feast apply.
  - Matérialiser les features dans l'online store.
  - Effectuer un offline retrieval simple pour visualiser les features récupérées.
- Le but : connecter l'ingestion (TP2) au futur pipeline d'entraînement (TP4).